16 | 分布式训练：如何加速你的模型训练？

2021-11-17 方远

《PyTorch深度学习实战》 课程介绍 >

讲述：方远
时长 16:18 大小 14.93M

你好，我是方远。

在之前的课程里，我们一起学习了深度学习必备的内容，包括构建网络、损失函数、优化方法等，这些环节掌握好了，我们就可以训练很多场景下的模型了。

但是有的时候，我们的模型比较大，或者训练数据比较多，训练起来就会比较慢，该怎么办呢？这时候牛气闪闪的分布式训练登场了，有了它，我们就可以极大地加速我们的训练过程。

这节课我就带你入门分布式训练，让你吃透分布式训练的工作原理，最后我还会结合一个实战项目，带你小试牛刀，让你在动手过程中加深对这部分内容的理解。

分布式训练原理

在具体介绍分布式训练之前，我们需要先简要了解一下为什么深度学习要使用 GPU。

在我们平时使用计算机的时候，程序都是将进程或者线程的数据资源放在内存中，然后在 CPU 进行计算。通常的程序中涉及到了大量的 if else 等分支逻辑操作，这也是 CPU 所擅长的计算方式。

而在深度学习中，模型的训练与计算过程则没有太多的分支，基本上都是矩阵或者向量的计算，而这种暴力又单纯的计算形式非常适合用 GPU 处理，GPU 的整个处理过程就是一个流式处理的过程。

但是再好的车子，一个缸的发动机也肯定比不过 12 个缸的，同理单单靠一个 GPU，速度肯定还是不够快，于是就有了多个 GPU 协同工作的办法，即分布式训练。分布式训练，顾名思义就是训练的过程是分布式的，重点就在于后面这两个问题：

1. 谁分布了？答案有两个：数据与模型。

2. 怎么分布？答案也有两个：单机多卡与多机多卡。

也就是说，为了实现深度学习的分布式训练，我们需要采用单机多卡或者多机多卡的方式，让分布在不同 GPU 上的数据和模型协同训练。那么接下来，我们先从简单的单机单卡入手，了解一下 GPU 的训练过程。

单机单卡

想象一下，如果让你把数据或者模型推送到 GPU 上，需要做哪几步操作呢？让我们先从单 GPU 的情况出发。

第一步，我们需要知道手头有多少 GPU。PyTorch 中使用 torch.cuda.is_available() 函数来判断当前的机器是否有可用的 GPU，而函数 torch.cuda.device_count() 则可以得到目前可用的 GPU 的数量。

第二步，获得 GPU 的一个实例。例如下面的语句：

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```
复制代码

这里 torch.device 代表将 torch.Tensor 分配到的设备，是一个设备对象实例，也就是 GPU。其中 cuda: 0 表示我们使用的是第一块 GPU。当然你也可以不用声明 “:0” ，默认就从第一块开始。如果没有 GPU（torch.cuda.is_available()），那就只能使用 CPU 了。

第三步，将数据或者模型推到 GPU 上去，这个过程我们称为**迁移**。

在 PyTorch 中，这个过程的封装程度非常高，换句话说，我们只需要保证即将被推到 GPU 的内容是张量（Tensor）或者模型（Module），就可以用 to() 函数快速进行实现。例如：

复制代码

```
data = torch.ones((3, 3))
print(data.device)
# Get: cpu

# 获得device
device = torch.device("cuda: 0")

# 将data推到gpu上
data_gpu = data.to(device)
print(data_gpu.device)
# Get: cuda:0

```

在上面这段代码中，我们首先创建了一个常规的张量 data，通过 device 属性，可以看到 data 现在是在 CPU 上的。随后，我们通过 to() 函数将 data 迁移到 GPU 上，同样也能通过 device 属性看到 data 确实已经存在于 GPU 上了。

那么对于模型，是否也是一样的操作呢？答案是肯定的，我们接下来看一个例子：

复制代码

```
net = nn.Sequential(nn.Linear(3, 3))
net.to(device)
```

这里仍旧使用 to() 函数即可。

单机单卡的模式，相当于有一批要处理加工的产品，只分给了一个工人和一台机器来完成，这种情况下数量少了还可以，但是一旦产品太多了，就得加人、加机器才能快速交工了。

深度学习也是一样，在很多场景中，比如推荐算法模型、语言模型等，往往都有着百万、千万甚至上亿的训练数据，这样如果只用一张卡的话肯定是搞不定了。于是就有了单机多卡和多机多卡的解决方案。

单机多卡

那么，在 PyTorch 中，单机多卡的训练是如何进行的呢？其实 PyTorch 提供了好几种解决方案，咱们先看一个最简单也是最常用的办法：nn.DataParallel()。其具体定义如下：

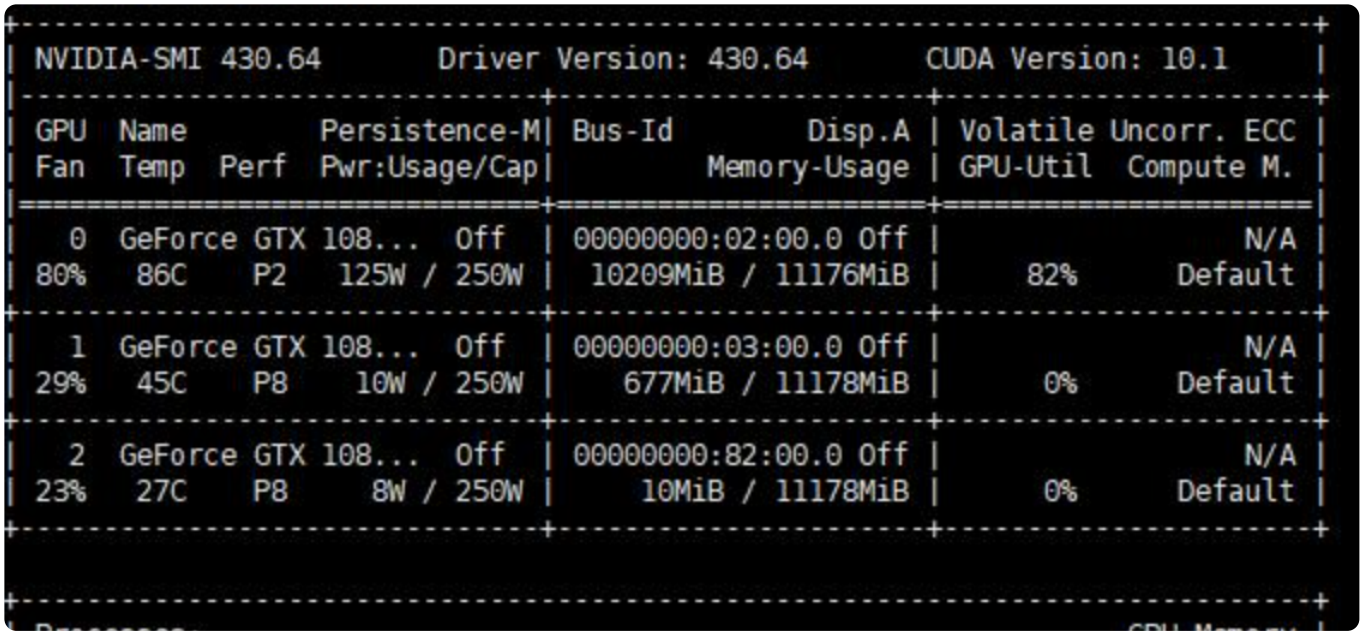
复制代码

```
1 torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)
```

在这里，module 就是你定义的模型，device_ids 即为训练模型时用到的 GPU 设备号，output_device 表示输出结果的 device，默认为 0 也就是第一块卡。

我们可以使用 nvidia-smi 命令查看 GPU 使用情况。如果你足够细心就会发现，使用多个卡做训练的时候，output_device 的卡所占的显存明显大一些。

继续观察你还会发现，使用 DataParallel 时，数据的使用是并行的，每张卡获得的数据都一样多，但是输出的 loss 则是所有的卡的 loss 都会在第 output_device 块 GPU 进行计算，这导致了 output_device 卡的负载进一步增加。


NVIDIA-SMI 430.64 Driver Version: 430.64 CUDA Version: 10.1
GPU Name Persistence-M Bus-Id Disp.A Volatile Uncorr. ECC
Fan Temp Perf Pwr:Usage/Cap Memory-Usage GPU-Util Compute M.
0 GeForce GTX 108... Off 00000000:02:00.0 Off N/A
80% 86C P2 125W / 250W 10209MiB / 11176MiB 82% Default
1 GeForce GTX 108... Off 00000000:03:00.0 Off N/A
29% 45C P8 10W / 250W 677MiB / 11178MiB 0% Default
2 GeForce GTX 108... Off 00000000:82:00.0 Off N/A
23% 27C P8 8W / 250W 10MiB / 11178MiB 0% Default

就这么简单？对，就这么简单，只需要一个 DataParallel 函数就可以将模型分发到多个 GPU 上。但是我们还是需要了解这内部的运行逻辑，因为只有了解了这个逻辑，在以后的开发中遇到了诸如**时间计算、资源预估、优化调试问题**的时候，你才可以更好地运用 GPU，让多 GPU 的优势真正发挥出来。

在模型的前向计算过程中，数据会被划分为多个块，被推送到不同的 GPU 进行计算。但是不同的是，模型在每个 GPU 中都会复制一份。我们看一下后面的代码：

复制代码

```
class ASimpleNet(nn.Module):
    def __init__(self, layers=3):
        super(ASimpleNet, self).__init__()
        self.linears = nn.ModuleList([nn.Linear(3, 3, bias=False) for i in ran
    def forward(self, x):
        print("forward batchsize is: {}".format(x.size()[0]))
        x = self.linears(x)
        x = torch.relu(x)
        return x

batch_size = 16
inputs = torch.randn(batch_size, 3)
labels = torch.randn(batch_size, 3)
inputs, labels = inputs.to(device), labels.to(device)
net = ASimpleNet()
net = nn.DataParallel(net)
net.to(device)
print("CUDA_VISIBLE_DEVICES :{}".format(os.environ["CUDA_VISIBLE_DEVICES"]))

for epoch in range(1):
    outputs = net(inputs)
```

```
# Get:
# CUDA_VISIBLE_DEVICES : 3, 2, 1, 0
# forward batchsize is: 4
# forward batchsize is: 4
# forward batchsize is: 4
# forward batchsize is: 4
```

在上面的程序中，我们通过 CUDA_VISIBLE_DEVICES 得知了当前程序可见的 GPU 数量为 4，而我们的 batch size 为 16，输出每个 GPU 上模型 forward 函数内部的 print 内容，验证了每个 GPU 获得的数据量都是 4 个。这表示，DataParallel 会自动帮我们将数据切分、加载到相应 GPU，将模型复制到相应 GPU，进行正向传播计算梯度并汇总。

多机多卡

多机多卡一般都是基于集群的方式进行大规模的训练，需要涉及非常多的方面，咱们这节课只讨论最基本的原理和方法。在具体实践中，你可能还会遇到其它网络或环境等问题，届时需要具体问题具体解决。

DP 与 DDP

刚才我们已经提到，对于单机多卡训练，有一个最简单的办法：DataParallel。其实 PyTorch 的数据并行还有一个主要的 API，那就是 DistributedDataParallel。而 **DistributedDataParallel 也是我们实现多机多卡的关键 API**。

DataParallel 简称为 DP，而 DistributedDataParallel 简称为 DDP。我们来详细看看 DP 与 DDP 的区别。

先看 DP，DP 是单进程控制多 GPU。从之前的程序中，我们也可以看出，DP 将输入的一个 batch 数据分成了 n 份（n 为实际使用的 GPU 数量），分别送到对应的 GPU 进行计算。

在网络前向传播时，模型会从主 GPU 复制到其它 GPU 上；在反向传播时，每个 GPU 上的梯度汇总到主 GPU 上，求得梯度均值更新模型参数后，再复制到其它 GPU，以此来实现并行。

由于主 GPU 要进行梯度汇总和模型更新，并将计算任务下发给其它 GPU，所以主 GPU 的负载与使用率会比其它 GPU 高，这就导致了 GPU 负载不均衡的现象。

再说说 DDP，DDP 多进程控制多 GPU。系统会为每个 GPU 创建一个进程，不再有主 GPU，每个 GPU 执行相同的任务。DDP 使用分布式数据采样器（DistributedSampler）加载数据，确保数据在各个进程之间没有重叠。

在反向传播时，各 GPU 梯度计算完成后，各进程以广播的方式将梯度进行汇总平均，然后每个进程在各自的 GPU 上进行梯度更新，从而确保每个 GPU 上的模型参数始终保持一致。由于无需在不同 GPU 之间复制模型，DPP 的传输数据量更少，因此速度更快。

DistributedDataParallel 既可用于单机多卡也可用于多机多卡，它能够解决 DataParallel 速度慢、GPU 负载不均衡等问题。因此，官方更推荐使用 DistributedDataParallel 来进行分布式训练，也就是接下来要说的 DDP 训练。

DDP 训练

DistributedDataParallel 主要是为多机多卡而设计的，不过单机上也同样可以使用。

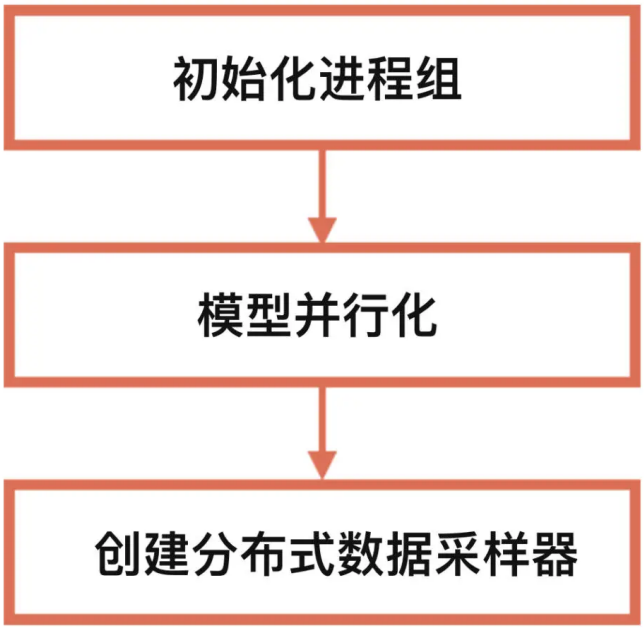
想要弄明白 DPP 的训练机制，我们先要弄明白这几个分布式中的概念：

group：即进程组。默认情况下，只有一个组，即一个 world。

world_size ：表示全局进程个数。

rank：表示进程序号，用于进程间通讯，表示进程优先级。rank=0 的主机为主节点。

使用 DDP 进行分布式训练的具体流程如下。接下来，我们就按步骤分别去实现。



第一步，初始化进程组。我们使用 init_process_group 函数来进行分布式初始化，其定义如下：

```
1 torch.distributed.init_process_group(backend, init_method=None,, world_size=-1
```

复制代码

我们分别看看定义里的相关参数：

backend：是通信所用的后端，可以是“nccl”或“gloo”。一般来说，nccl 用于 GPU 分布式训练，gloo 用于 CPU 进行分布式训练。

init_method：字符串类型，是一个 url，用于指定进程初始化方式，默认是“env://”，表示从环境变量初始化，还可以使用 TCP 的方式或共享文件系统 。

world_size：执行训练的所有的进程数，表示一共有多少个节点（机器）。

rank：进程的编号，也是其优先级，表示当前节点（机器）的编号。

group_name：进程组的名字。

使用 nccl 后端的代码如下。

```
1 torch.distributed.init_process_group(backend="nccl")
```

复制代码

完成初始化以后，第二步就是模型并行化。正如前面讲过的，我们可以使用 DistributedDataParallel，将模型分发至多 GPU 上，其定义如下：

```
复制代码
1 torch.nn.parallel.DistributedDataParallel(module, device_ids=None, output_devi
```

DistributedDataParallel 的参数与 DataParallel 基本相同，因此模型并行化的用法只需将 DataParallel 函数替换成 DistributedDataParallel 即可，具体代码如下。

```
复制代码
1 net = torch.nn.parallel.DistributedDataParallel(net)
```

最后就是创建分布式数据采样器。在多机多卡情况下，分布式训练数据的读取也是一个问题，不同的卡读取到的数据应该是不同的。

DP 是直接将一个 batch 的数据划分到不同的卡，但是多机多卡之间进行频繁的数据传输会严重影响效率，这时就要用到分布式数据采样器 DistributedSampler，它会为每个子进程划分出一部分数据集，从而使 DataLoader 只会加载特定的一个子数据集，以避免不同进程之间有数据重复。

创建与使用分布式数据采样器的代码如下。

```
复制代码
1 train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
2 data_loader = DataLoader(train_dataset, batch_size=batch_size, sampler=train_s
```

结合代码我给你解读一下。

首先，我们将 train_dataset 送到了 DistributedSampler 中，并创建了一个分布式数据采样器 train_sampler。

然后在构造 DataLoader 的时候，参数中传入了一个 sampler=train_sampler，即可让不同的进程节点加载属于自己的那份子数据集。也就是说，使用 DDP 时，不再是从主 GPU 分发数据到其他 GPU 上，而是各 GPU 从自己的硬盘上读取属于自己的那份数据。

为什么要使用分布式训练以及分布式训练的原理我们就讲到这里。相信你已经对数据并行与模型并行都有了一个初步的认识。

小试牛刀

下面我们将会讲解一个官方的 ImageNet 的示例，以后你可以把这个小项目当做分布式训练的一个模板来使用。

这个示例可对使用 DP 或 DDP 进行选配，下面我们就一起来看核心代码。

复制代码

```
if args.distributed:
    if args.dist_url == "env://" and args.rank == -1:
        args.rank = int(os.environ["RANK"])
    if args.multiprocessing_distributed:
        # For multiprocessing distributed training, rank needs to be the
        # global rank among all the processes
        args.rank = args.rank * ngpus_per_node + gpu
    dist.init_process_group(backend=args.dist_backend, init_method=args.dist_
                            world_size=args.world_size, rank=args.rank)
```

这里你可以重点关注示例代码中的“args.distributed”参数，args.distributed 为 True，表示使用 DDP，反之表示使用 DP。

我们来看 main_worker 函数中这段针对 DDP 的初始化代码，如果使用 DDP，那么使用 init_process_group 函数初始化进程组。ngpus_per_node 表示每个节点的 GPU 数量。

我们再来看 main_worker 函数中的这段逻辑代码。

复制代码

```
if not torch.cuda.is_available():
    print('using CPU, this will be slow')
elif args.distributed:
    # For multiprocessing distributed, DistributedDataParallel constructor
```

```
    # should always set the single device scope, otherwise,
    # DistributedDataParallel will use all available devices.
    if args.gpu is not None:
        torch.cuda.set_device(args.gpu)
        model.cuda(args.gpu)
        # When using a single GPU per process and per
        # DistributedDataParallel, we need to divide the batch size
        # ourselves based on the total number of GPUs we have
        args.batch_size = int(args.batch_size / ngpus_per_node)
        args.workers = int((args.workers + ngpus_per_node - 1) / ngpus_per_nod
        model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[a
    else:
        model.cuda()
        # DistributedDataParallel will divide and allocate batch_size to all
        # available GPUs if device_ids are not set
        model = torch.nn.parallel.DistributedDataParallel(model)
elif args.gpu is not None:
    torch.cuda.set_device(args.gpu)
    model = model.cuda(args.gpu)
else:
    # DataParallel will divide and allocate batch_size to all available GPUs
    if args.arch.startswith('alexnet') or args.arch.startswith('vgg'):
        model.features = torch.nn.DataParallel(model.features)
        model.cuda()
    else:
        model = torch.nn.DataParallel(model).cuda()
```

这段代码是对使用 CPU 还是使用 GPU、如果使用 GPU，是使用 DP 还是 DDP 进行了逻辑选择。我们可以看到，这里用到了 DistributedDataParallel 函数或 DataParallel 函数，对模型进行并行化。

并行化之后就是创建分布式数据采样器，具体代码如下。

复制代码

```
if args.distributed:
    train_sampler = torch.utils.data.distributed.DistributedSampler(train_data
else:
    train_sampler = None

train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=args.batch_size, shuffle=(train_sampler is None)
    num_workers=args.workers, pin_memory=True, sampler=train_sample
```

这里需要注意的是，**在建立 Dataloader 的过程中，如果 sampler 参数不为 None，那么 shuffle 参数不应该被设置。**

最后，我们需要为每个机器节点上的每个 GPU 启动一个进程。PyTorch 提供了 torch.multiprocessing.spawn 函数，来在一个节点启动该节点所有进程，具体的代码如下。

复制代码

```
ngpus_per_node = torch.cuda.device_count()
if args.multiprocessing_distributed:
    # Since we have ngpus_per_node processes per node, the total world_size
    # needs to be adjusted accordingly
    args.world_size = ngpus_per_node * args.world_size
    # Use torch.multiprocessing.spawn to launch distributed processes: the
    # main_worker process function
    mp.spawn(main_worker, nprocs=ngpus_per_node, args=(ngpus_per_node, args))
else:
    # Simply call main_worker function
    main_worker(args.gpu, ngpus_per_node, args)
```

对照代码我们梳理一下其中的要点。之前我们提到的 main_worker 函数，就是每个进程中，需要执行的操作。ngpus_per_node 是每个节点的 GPU 数量（每个节点 GPU 数量相同），如果是多进程，ngpus_per_node * args.world_size 则表示所有的节点中一共有多少个 GPU，即总进程数。

一般情况下，进程 0 是主进程，比如我们会在主进程中保存模型或打印 log 信息。

当节点数为 1 时，实际上就是单机多卡，所以说 DDP 既可以支持多机多卡，也可以支持单机多卡。

main_worker 函数的调用方法如下。

复制代码

```
main_worker(args.gpu, ngpus_per_node, args)
```

其中，args.gpu 表示当前所使用 GPU 的 id。而通过 mp.spawn 调用之后，会为每个节点上的每个 GPU 都启动一个进程，每个进程运行 main_worker(i, ngpus_per_node, args)，其中 i 是从 0 到 ngpus_per_node-1。

模型保存的代码如下。

复制代码

```
if not args.multiprocessing_distributed or (args.multiprocessing_distributed
        and args.rank % ngpus_per_node == 0):
    save_checkpoint({
        'epoch': epoch + 1,
        'arch': args.arch,
        'state_dict': model.state_dict(),
        'best_acc1': best_acc1,
        'optimizer' : optimizer.state_dict(),
    }, is_best)
```

这里需要注意的是，使用 DDP 意味着使用多进程，如果直接保存模型，每个进程都会执行一次保存操作，此时只使用主进程中的一个 GPU 来保存即可。

好，说到这，这个示例中有关分布式训练的重点内容我们就讲完了。

小结

恭喜你走到这里，这节课我们一起完成了分布式训练的学习，最后咱们一起做个总结。

今天我们不但学习了为什么要使用分布式训练以及分布式训练的原理，还一起学习了一个分布式训练的实战项目。

在分布式训练中，主要有 DP 与 DDP 两种模式。其中 DP 并不是完整的分布式计算，只是将一部分计算放到了多张 GPU 卡上，在计算梯度的时候，仍然是“一卡有难，八方围观”，因此 DP 会有负载不平衡、效率低等问题。而 DDP 刚好能够解决 DP 的上述问题，并且既可以用于单机多卡，也可以用于多机多卡，因此它是更好的分布式训练解决方案。

你可以将今天讲解的示例当做分布式训练的一个模板来使用。它包括了 DP 与 DPP 的完整使用过程，并且包含了如何在使用 DDP 时保存模型。不过这个示例中的代码里其实还有更多的细节，建议你留用课后空余时间，通过精读代码、查阅资料，多动手、多思考来巩固今天的学习成果。

每课一练

在 torch.distributed.init_process_group(backend=“nccl”) 函数中，backend 参数可选哪些后端，它们分别有什么区别？

推荐你好好研读今天的分布式训练 demo，也欢迎你记录自己的学习感悟或疑问，我在留言区等你。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | 可视化工具：如何实现训练的可视化监控？

下一篇 17 | 图像分类（上）：图像分类原理与图像分类模型

精选留言 (4)

写留言

小门神
2021-11-18

老师，您好，请问利用pytorch对模型进行分布式训练后，在对模型进行推理时的处理和单

机单卡有区别吗

展开

作者回复: 你好，感谢你的留言。没有区别^^

1

马克图布

2021-11-17

问：在 `torch.distributed.init_process_group(backend="nccl")` 函数中，backend 参数可选哪些后端，它们分别有什么区别？

答：根据文档，可选后端有 `gloo`, `nccl` 和 `mpi`。区别如下。

...

展开

作者回复: ^^，一万个赞，厉害。

马克图布

2021-11-17

问题：

train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
data_loader = DataLoader(dataset, batch_size=batch_size, sampler=train_sampler)

...

展开

编辑回复: 已修改，刷新可见。

hsiang271828

2021-11-17

你好，我看到官方的 ImageNet 的示例中299行有如下代码：

```
# compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()
```

optimizer.step()…

展开

作者回复: 你好，hsiang271828，感谢你的留言。

如果1个优化器对应1个模型的时候(1个优化器可以对应多个模型)。optimizer.zero_grad()与model.zero_grad()是相同的。

如果是多个模型的时候，就要具体情况具体分析了。如果直接optimizer.zero_grad()的话，就会把所有模型的梯度清零。